Science of Computer Programming 8 (1987) 43–67
North-Holland

PROOF RULES FOR FAULT TOLERANT DISTRIBUTED PROGRAMS

Mathai JOSEPH*
Computer Science Group, Tata Institute of Fundamental Research, Bombay 400005, India
Abha MOITRA
Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, U.S.A.
Neelam SOUNDARARAJAN**
Department of Computer & Information Science, Ohio State University, Columbus, OH 43210, U.S.A.

Communicated by K. R. Apt
Received October 1984
Revised May 1986

Abstract. Proving the properties of a program which must execute on a distributed system whose nodes may fail is a complex task. Such proofs must take into account the effects of hardware failures at all possible points in the execution of individual processes. The difficulty in accomplishing this is compounded by the need to cater also for the simultaneous failure of two or more processing nodes. In this paper, we consider programs written in a version of Hoare's CSP and define a set of axioms and inference rules by which proofs can be constructed in three steps: proving the properties of each process when its communicants are prone to failure, establishing the effects of failure of each process, and combining these two steps to determine the fault tolerant properties of the whole program. The proof system is thus compositional, in the sense that proofs can be constructed in a modular way.

1. Introduction

In some earlier work [3, 4], we had studied the problem of proving the properties of distributed programs which execute on hardware that may fail. For a restricted class of programs, and for some kinds of hardware failures, it was shown that the properties of such programs can be proved and that these proofs can be used to restructure programs for improved resilience to hardware failures. Programs were written in a slightly extended version of CSP [2] and the proof system of Apt, Francez and de Roever [1] was used under the assumption that failures in a processor or in a communication channel could be detected, and that recovery could be initiated by 're-making' the process, causing it to 'repair' its channels and resume execution from its initial state. In general, recovery required the cooperative action of several processes.

* Present address: Department of Computer Science, University of Warwick, Coventry CV4 7AL United Kingdom.

** Part of this work was supported by NSF Grant ECS-8404725.

As earlier, we are more concerned with proof techniques for fault tolerant programs than with the wide variety of faults that may in practice occur. Briefly, the 'faults' we are concerned with in this paper are those that can be attributed to hardware failures, and we call a program 'fault tolerant' if it can be proved to produce correct behaviour in spite of the occurrence of such failures. A necessary prerequisite for this is to assume that hardware failures are detected, and it is well known that this is not always possible. To simplify the discussion, we shall only consider certain kinds of hardware failures: for example, we shall assume that communication between processes is reliable and that failures are not malicious. When we refer to a process 'failing', we shall mean that it has been subject to a detected hardware failure; 'recovery' of a process will mean restarting its execution from some previous state. In fact, we shall assume that processes resume execution from their initial state, since that is most easily possible. In general, we have chosen to make very simple assumptions about hardware failures so that attention can be concentrated on proof techniques for fault tolerant programs.

Apart from their use in proving properties of fault tolerant programs, local and global invariants can be analyzed to provide guidelines for constructing fault tolerant versions of correct distributed programs [4]. Such guidelines are quite useful for simple programs, but we have found that they do not always provide enough information to help in constructing larger and more complex fault tolerant programs. Informally, we can say that local and global invariants allow characterization of the 'low level' behaviour of a program and that this is sometimes inadequate when 'high level' properties of a fault tolerant program must be established; for example, a global invariant provides links between communication commands in pairs of processes but it is difficult to use it to both represent the effects of failure in one process on other processes and account for the simultaneous failure of more than one process.

Consider a program S_1 in which process R_1 sends process R_2 an ascending sequence of integers and R_2 sums successive pairs of integers and sends the results to process R_3 to be printed:

$$
\begin{aligned}
S_1 :: [\, & R_1 :: i := 1; *[\{LI(R_1)\}R_2 ! i \rightarrow i := i+1] \\
& \| \\
& R_2 :: R_1 ? x; *[\{LI(R_2)\}R_1 ? y \rightarrow R_3 ! (x+y); R_1 ? x] \\
& \| \\
& R_3 :: *[\{LI(R_3)\}R_2 ? t \rightarrow \text{“print } t\text{”}] \\
&]
\end{aligned}
$$

Assume process R_2 fails (and recovers) and that the fact that this failure has occurred is registered by R_1. What actions are then possible for R_1 so that the program overcomes the effect of the failure? If R_1 registers the failure when $i > 2$ and i is even, the local invariants $LI(R_1)$, $LI(R_2)$ and $LI(R_3)$ and a suitable global invariant will permit the inference that the sum of the two integers $(i-3)$ and $(i-2)$ has been

sent by R_2 to R_3. But if the value of i is odd, no assertion can be made about whether R_2 failed after or before it communicated the new sum to R_3. In a fault tolerant version of the program, R_1 must then assume the worst and decrement i by 2 before re-sending values to R_2. And this can lead to the same value being printed twice (or more often, if there are repeated failures in R_2). Unfortunately, even with as few as three processes, the reasoning that leads to this conclusion must be based on properties of the whole program and the assertion describing this condition is no longer simple. It can easily be seen that the situation can get rapidly out of hand as the complexity of each process and/or the number of processes increases.

If R_3 is modified to print only those values received from R_2 that are in strictly ascending order, such repetitions can be avoided. But if R_2 and R_3 fail simultaneously, i.e. within the same interval of time, it can no longer be assured that values are printed in this order and we must accept a weaker global invariant that admits the possibility of repetitions. In general, in all such cases we must either rely on the intuition of the programmer in specifying these effects and their combinations correctly, or look for a methodology by which the effects can be systematically derived. Even a limited acquaintance with the construction of fault tolerant programs will show that intuition, by itself, is not sufficient: we have found building such programs and proving their properties to be an extremely complex task unless it is supported by a sound methodology.

We have therefore been investigating other techniques for constructing fault tolerant programs. A more promising approach than dealing with the program as a whole would be to decompose the main problem into the sub-problems of:

(a) Obtaining the behaviour of each process P_i allowing for failures (and subsequent recoveries) of its communicants; however, in this step, we do *not* consider failures of the process P_i itself.

(b) Obtaining, from (a), the behaviour of P_i, now allowing for its own failures.

Notation: We shall denote by P_i the execution of the ith process without any failures, and, by $[P_i]$ the execution of the same process in the presence of failures. Thus, in Step (a) we obtain the behaviour of P_i, and in (b) we obtain the behaviour of $[P_i]$ from that of P_i.

(c) Combining the behaviours of $[P_i]$, $i = 1, \ldots, n$, to obtain the behaviour of the program as a whole.

Steps (a) and (b) can then be performed locally (i.e., in isolation) for each process and Step (c) requires the use of proof rules for communication between processes. Steps (a), (b) and (c) are handled in Sections 5, 6 and 7 respectively. It should be noted that the properties of the same process with and without failures can be rather different and this should emerge from the application of Step (b), rather than from informal reasoning by the programmer.

The CSP proof system of Soundararajan [8] follows similar steps in reasoning about programs: properties of individual processes are proved in isolation and then combined by a rule of parallel composition where the post-condition of the individual

processes, along with the requirement of mutual compatibility between the record of the sequences of communication of the various processes, allows us to draw appropriate conclusions about the whole CSP program. We shall extend the axioms and inference rules of this proof system for our purposes, adding clauses to account for detected failures, and then prove the fault tolerant properties of a simple program.

Our model of fault tolerant programs is rather simple: we do not, for instance, consider the possibility of malicious behaviour. The purpose of this paper is not so much to propose an elaborate model of fault tolerant programs that includes all of the problems that might arise in real programs, as to present an axiomatic approach that may be used for analysing fault tolerant programs. Naturally, the approach can be extended to deal with more general classes of fault tolerant programs than those considered here.

2. A system model

Assume every process in a CSP program executes on a separate processing node and that the nodes are interconnected by a communication medium. A processing node has some private memory, inaccessible to other nodes, and two or more processors which separately execute the code for the process and deposit the results in this memory. A fault occurs in a node when there is some hardware failure in a processor: it is assumed that this can be detected as a discrepancy between the actions of the processors at the node. These are the *only* faults considered and it is assumed that there are no communication failures. In general, we shall not consider conspiracies and malicious failures, though some of the assumptions we make are valid only if appropriate solutions are used for such failures [6].

When there are two processors in a node, the hardware faults we consider can be detected by 'matching' circuits; if there are three or more processors, simple majority logic can be used. A node with these properties is very similar to the 'failstop' processors of Schlichting and Schneider [7] which perform a consistency check after each instruction and stop when a hardware error is detected. However, for our purposes, the processors need not execute in step as it is only necessary that they synchronize at selection points (so that all the processors make the same choice) and that faults be checked for periodically and before any communication takes place with another node. When a fault is detected in a node, execution of the resident process must cease and the node fails to respond to any communication requests. A failed node (and its resident process) is said to be *withdrawn.*

We can summarize the assumptions made thus far as follows:

Assumption 1. There are no faults in the communication pathways between processes: a message sent correctly will be received correctly some finite time later and two successive messages sent from one process to another will be received in the order that they were sent.

Assumption 2. All hardware faults in a node are detected and cause execution of the node to stop before further communication takes place with any other node.

Assumption 3. A process on a fault free node which attempts to communicate with another process on a withdrawn node will receive an error signal, so failure of a node can be detected by attempts at communication from other nodes.

Without making any further assumptions about the communication patterns in the program (as were made, for example, in [3]), these assumptions are sufficient to guarantee detection of each node failure if no more than one node fails within the time required for this failure to be detected by some other node. But there are several reasons why this is an unreasonable constraint. For example, we shall wish to make assertions about the final values of the variables of a process, and that is only possible if the process has not failed after its last communication. And for arbitrary communications graphs, or multiple simultaneous failures, some more assumptions are in any case needed. Since any of the processes of a program may suffer as a result of a node failure, assume there is a separate communication checker process whose function is to detect all node failures.

Assumption 4. A communication checker process P_0 executes on a separate node and regularly interrogates each node and detects a failure by the lack of a response.

The communication checker has an additional function: when it detects a failure of a node, it 're-makes' the process and the process resumes execution from its initial state; if more than one initial state is possible then it resumes in any of those initial states. (It is possible that the process re-starts execution on a new node—assuming adequate hardware redundancy. In this case, P_0 will also have to maintain a table indicating the process running on each node, and which nodes are 'up' and free.) When a process P_j is re-made, any attempt by another process P_i at communication with P_j will result in P_i first receiving a message to the effect that P_j has failed and recovered (been re-made) since their last communication, so that P_i can, perhaps, re-send some of the information that it had previously sent to P_j. (In our system model, P_i would then need to discover the identity of the new node at which P_j is running; being an implementation detail, this will be transparent to the programmer, and will not be explicitly represented in our axiomatics.) To complete the model, we must assume that there are a number of communication checkers (one master and several standbys, for example) each in communication with the others so that failure of a communication checker does not compromise the reliability of the whole system.

Assumption 5. At least one communication checker is executing correctly at all times.

It must be emphasized that the system model outlined here has been kept as simple as possible because the purpose of this paper is really to construct proof

techniques that can be used to examine the effects of hardware failures on distributed programs. Nevertheless, it is useful to know that a number of practical implementations of such a model are possible, e.g. on a shared memory system, or on a local area network where the communication interface of each node stores the identity of the process executing at the node. A generous amount of hardware redundancy is needed, but its actual extent will depend on the degree of failure resilience required.

3. Fault detection and recovery

We extend the communication statements of CSP to allow us to deal with failures. Using the symbol '.' to mean either input ('?') or output ('!'), a statement in process P'_1 to communicate with process P'_2 appears as

$$P'_2 \,.\, x\langle\langle S'\rangle\rangle \tag{1}$$

and in a guarded command as

$$b;\ P'_2 \,.\, x \rightarrow S\langle\langle S'\rangle\rangle \tag{2}$$

where b is a boolean expression.

As in CSP, a communication command such as $P'_2 \,.\, x$ may be selected for execution in two ways: deterministically, as in (1), and non-deterministically (subject to b evaluating to true), as in (2). When such a statement is selected for execution in a correct program, there are two possibilities:

(a) Process P'_2 may be ready to reciprocate the communication, and the command $P'_2 \,.\, x$ is executed;

(b) Process P'_2 may have failed since it last communicated with P'_1, and the statement $\langle\langle S'\rangle\rangle$ is executed: if P'_2 is still in a failed state, it is implicitly re-made before S' is executed.

Note that exactly one of these possibilities will be selected: either $P'_2 \cdot x$ is correctly executed, or $\langle\langle S'\rangle\rangle$ is executed. In our further discussion, we shall call $\langle\langle S'\rangle\rangle$ the 'fault' alternative. To keep the control flow simple, we do not allow any communication statement to be included in a fault alternative.

To illustrate the use of such statements, consider the following example of the well-known bounded buffer. A producer process P'_1 sends a sequence of numbered lines to a process P'_2 with a buffer of size n. The buffer process in turn sends a sequence of lines to the consumer process P'_3 which sends each line to a printer process P'_4:

$S_2 :: [P'_1 \| P'_2 \| P'_3 \| P'_4]$

$P'_1 ::$ *pseq*: integer; *ready*: boolean; *nextline*: line;

pseq := 0; *ready* := *false*;

$*[\neg ready \rightarrow nextline := Line(pseq+1);\ ready := true$

$\square$

$ready;\ P_2'!(nextline, pseq+1) \rightarrow pseq := pseq+1;\ ready := false$

$]$

$P_2'::$ *in*, *out*: integer;

$A:[0..(n-1)]$ of record *ln*: line; *linenum*: integer end;

$in := 0;\ out := 0;$
$*[in < out + n - 1;\ P_1'?A[in \bmod n] \rightarrow in := in + 1$

$\square$

$out < in;\ P_3'!A[out \bmod n] \rightarrow out := out + 1$

$]$

$P_3'::$ *ln*: line; *num*: integer;
$*[P_2'?ln, num \rightarrow P_4'!ln$

$]$

$P_4'::$ *ln*: line;

$*[P_3'?ln \rightarrow skip$

$]$

Let us assume for simplicity that the producer process, P_1', and the printer process, P_4', never fail. A failure in the buffer process, P_2', will be detected by P_1' and P_3' and will cause the loss of $(out - in)$ lines. As *out* and *in* are local variables of P_2', it must then be assumed by P_1' that up to n lines may be lost. Thus, a solution is for P_1' to re-send these lines to P_2'. If, in fact, $(out - in)$ were less than n at the time of failure, this may lead to up to n repetitions in lines sent to P_3'. To suppress such duplicate lines, P_3' can be altered to forward to P_4' only those lines that have numbers in strictly ascending order; e.g., if *lastnum* is the number of the last line printed, the next line to be printed must have $num > lastnum$. Failure of P_3' will be detected by P_2' and P_4' and may result in the loss of a line taken from P_2' but not yet printed. P_2' must therefore make provision for repeating the last line sent to P_3' and, if this line had been printed, this will lead to a duplicated line. The last case to be considered is when P_2' and P_3' fail within an interval of time such that the value of *lastnum* in P_3' is lost and duplicate lines sent from P_2' reach the printer.

This informal analysis suggests that failure of P_2' and P_3' will, in general, lead to duplicated printing. The extent of such duplication depends on the points of failure, on the number of individual failures in P_2' and P_3', and on the interaction between

the effects of failures in P_2' and P_3'. We shall later formally prove the fault tolerant properties of S_3, which is a fault tolerant version of S_2.

$S_3 :: [P_1 \| P_2 \| P_3 \| P_4]$

P_1 :: *pseq*, *sent* : integer; *ready* : boolean; *nextline* : line;

pseq := 0; *sent* := 0; *ready* := *false*;
* [¬*ready* → *nextline* := *Line*(*pseq* + 1); *ready* := *true*

□

ready; P_2 ! (*nextline*, *pseq* + 1) → *pseq* := *pseq* + 1; *ready* := *false*;

sent := *sent* + 1

《*ready* := *false*;

[*sent* < *n* → *pseq* := *pseq* − *sent*

□

sent ≥ *n* → *pseq* := *pseq* − *n*

]; *sent* := 0》

]

P_2 :: *in*, *out* : integer; *output* : boolean;

A : [0 . . (*n* − 1)] of record *ln* : line; *linenum* : integer end;

in := 0; *out* := 0; *output* := *false*;

* [*in* < *out* + *n* − 2; P_1 ? *A*[*in* mod *n*] → *in* := *in* + 1

□

out < *in*; P_3 ! *A*[*out* mod *n*] → *out* := *out* + 1; *output* := *true*

《[*output* → *out* := *out* − 1

□

¬*output* → *skip*

]; *output* := *false*》

]

P_3 :: *ln* : line; *num*, *lastnum* : integer;

lastnum := 0;

```
* [ P2 ? ln, num → [ num > lastnum → P4 ! ln; lastnum := num
                  □
                  num ≤ lastnum → skip
                ]
                ⟪skip⟫
  ]
P4 :: ln : line;
  * [ P3 ? ln → skip
            ⟪skip⟫
  ]
```

Note that the program takes into account the possibility of repeated failures of P_2 and P_3 at all points in their execution, including the cases where they fail more than once before engaging in any communication.

4. Communication sequences

Let every communication in an (extended) CSP program be characterized by a triple of the form $\langle i, j, m\rangle$, where i is an integer index for the sender process, j a similar index for the receiver process and m the communication (or message). Thus the communication resulting from the simultaneous execution of the statement P_3 ! 5 in process P_1 and P_1 ? x in process P_3 would be represented by the triple $\langle 1, 3, 5\rangle$. We shall refer to communications resulting from the execution of input and output commands as 'explicit' communications.

'Implicit' communication takes place without the execution of input and output commands. There are two kinds of implicit communication and we shall characterize them by 'messages' received implicitly by a process:

- δ received by P_0 when some process P_i fails,
- ρ received by a process when it attempts to communicate with a process that has failed and has been re-made since their last communication.

But it must be emphasized that implicit communications do not necessarily represent the transmission of real messages from a sender to a receiver; in particular, the message 'δ' originates from the detection of failure in a process by the communication checker process P_0, rather than from any action assumed to be taken by the failed process. When a process P_i fails (and is re-made) then before any other process P_j, $j \neq 0$, can start communication with P_i, P_j must receive the message 'ρ'.

With every process P_i of a CSP program, we associate a communication sequence h_i which consists of triples denoting communications sent or received by P_i. Thus

the execution of the command $P_3 ! 5$ in process P_1 and the command $P_1 ? x$ in process P_3 will result in two identical triples, both equal to $\langle 1, 3, 5\rangle$, being concatenated to the communication sequences of P_1 and P_3. A failure of a process P_i (or rather the detection of P_i's failure and its re-making by P_0) is recorded as $\langle i, 0, \delta\rangle$ in h_i; and before any other process P_j can communicate with P_i, P_j records (its observation of) this failure as $\langle i, j, \rho\rangle$.

The following operations are defined over sequences:

$\|h\|$	is the length of the sequence h; for the empty sequence ε the length is zero,
$h_1 \hat{} h_2$	concates h_2 to the end of h_1; we will also use $\hat{}$ to indicate concatenation of a sequence with an element and an element with a sequence. The usage will be clear from the context,
$h \mid i$	is the subsequence of all elements of h which are of the form $\langle i, j, m\rangle$ or $\langle j, i, m\rangle$,
$h[k]$	is the kth element of h from the beginning,
$h(j:k]$	is the subsequence of h from its jth element to its kth element,
$h_i \subseteq h_j$	if for $1 \leq k \leq \|h_i\|$, $h_i[k] = h_j[k]$.

Much of this has been taken from Soundararajan [8], except for the introduction of implicit communications. We can therefore use the general form of the axioms and rules of inference defined there, with adaptations to deal with the extensions to CSP described above.

5. Axioms and rules of inference

Hoare-style proof systems are characterized by triples of the form $\{p\}\ S\ \{q\}$, which are interpreted as saying that if the predicate p is true before the execution of S, then the predicate q will be true if and when execution of S is completed. Consider now the execution of the process P_5:

$$P_5 :: [true \rightarrow P_6 ! 1; *[true \rightarrow skip]$$
$$\square$$
$$true \rightarrow P_6 ! 2]$$

Assume that in an execution of P_5, the first guarded command is chosen in the alternative statement so that 1 is output to P_6 and P_5 then loops in the repetitive command. When considering P_5's normal execution (i.e. without faults), it would be correct to annotate P_5 with the assertions

$$\{h_5 = \varepsilon\}\ P_5\ \{h_5 = \langle\langle 5, 6, 2\rangle\rangle\}$$

because the post-condition is indeed valid for the only case when P_5 does terminate; when P_5 does not terminate, a partial-correctness proof system will allow any arbitrary post-condition to be asserted.

That is not the case for the execution of P_5 in an environment where faults may appear. Taking the execution of P_5 described above, assume P_5 fails when executing its repetitive command. When P_5 is re-made, let its execution be such that this time the second guard is chosen. The sequence h_5 will then consist of

$$\langle\langle 5, 6, 1\rangle\langle 5, 0, \delta\rangle\langle 5, 6, 2\rangle\rangle$$

The post-condition for $[P_5]$ (where $[P_5]$ denotes fault prone execution of P_5) must thus be such that it is satisfied by *any* sequence of partial executions each ending in failure followed by recovery, followed by one ending in termination.

Clearly then, it is not possible to obtain the post-condition of $[P_5]$ from the post-condition of P_5 alone: we need also to know the communications that P_5 can participate in along *all* paths, those that lead to normal termination (these will be the paths about which information will be available in the post-condition of P_5), and those that do not. One possible solution is to have not only the post-condition q_5 of P_5, but also the invariant r_5 that its communication sequence h_5 will satisfy at all times during its execution. r_5 will then tell us what values h_5 may have at any point along the terminating and non-terminating execution paths of P_5.

Given the invariant r_5, and the post-condition q_5 of P_5 (i.e. the failure free P_5), it is indeed possible to obtain the post-condition of $[P_5]$ (i.e. the failure prone execution of P_5) by noting that an execution of $[P_5]$ is, in fact, several partial executions of P_5, each ending in failure, followed by a final complete execution of P_5. Hence the value of h_5 when an execution of $[P_5]$ finishes will be the concatenation of the communication sequences corresponding to the partial executions and the final complete execution of P_5 (each separated from the next by $\langle 5, 0, \delta\rangle$ to record the failure and recovery of P_5). Each of these sequences will satisfy r_5 (the final sequence will also satisfy q_5—or rather the final sequence will have a value 'allowed by' q_5, since q_5 may refer not only to h_5, but also to the other variables of P_5).

We use the following notation to describe the invariant and the post-condition

$$(r)\ \{p\}\ S\ \{q\}$$

and to mean the following: if p and r are satisfied initially, then throughout the execution of statement S from P_i, the sequence h_i will satisfy r, and if and when S terminates, the predicate q will hold; r is a predicate over h_i only, and does not refer to any other variables of P_i.

This extension of the standard Hoare notation is needed to derive the behaviour of $[P_i]$ from that of P_i. In fact, once we have available the invariant r_i and the post-condition q_i of P_i, it becomes possible to obtain not only the post-condition q_i' of $[P_i]$, but also the invariant r_i' that h_i satisfies during the execution of $[P_i]$; this is particularly useful for dealing with non-terminating processes, since for such processes the post-condition (usually '*false*') does not really say anything about the behaviour. In summary, invariants are used to obtain the behaviour of the failure prone process from the behaviour of the failure free process; once invariants are present, they take on the additional role of characterizing non-terminating programs. (We shall return to non-terminating processes and programs in Section 8.)

We can now write down the rules of inference corresponding to the various statements that may appear in a process P_i $(i>0)$; these rules will allow us to prove results of the kind

$$(r_i)\ \{p_i\}\ P_i\ \{q_i\}$$

about the failure free process P_i. A final rule will then allow us to obtain the behaviour (expressed as)

$$(r_i')\ \{p_i\}\ [P_i]\ \{q_i'\}$$

of the failure prone execution $[P_i]$. This rule will just be a formalization of the relation between P_i and $[P_i]$ discussed in the last few paragraphs.

Notation: In the rest of the paper, assertions with primes—e.g. r', $r_1', \ldots, q'$, $q_1', \ldots$—will be associated with failure prone processes, and unprimed assertions with fault free processes.

The following axioms and rules of inference refer to statements executed in a process P_i whose communication sequence is denoted by h_i, $i>0$. Notice that we do not define the communication checker process P_0 as it is assumed to be part of the underlying implementation.

R1. *Skip*

$$(r)\ \{p\}\ skip\ \{p\}$$

R2. *Output*

$$\frac{p \Rightarrow q^{h}_{h_i \hat{}\langle i,j,e\rangle},\ q \Rightarrow r}{(r)\ \{p\}\ P_j\ !\ e\ \{q\}}$$

R3. *Input*

$$\frac{p \Rightarrow \forall m\ .\ q^{x,h_i}_{m,h_i \hat{}\langle j,i,m\rangle},\ q \Rightarrow r}{(r)\ \{p\}\ P_j\ ?\ x\ \{q\}}$$

R4. *Fault tolerant communication*

$$(r)\ \{p\}\ C_j\ \{q\} \text{ where } C_j \text{ is either } P_j\ ?\ x \text{ or } P_j\ !\ e$$

$$p \Rightarrow q1^{h_i}_{h_i \hat{}\langle j,i,p\rangle}$$

$$\frac{(r)\ \{q1\}\ S\ \{q\}}{(r)\ \{p\}\ C_j\langle\langle S\rangle\rangle\ \{q\}}$$

R5. *Assignment*

$$(r)\ \{p^{x}_{e}\}\ x := e\ \{p\}$$

R6. *Composition*

$$\frac{(r)\ \{p\}\ S_1\ \{q1\},\ (r)\ \{q1\}\ S_2\ \{q\}}{(r)\ \{p\}\ S_1;\ S_2\ \{q\}}$$

R7. *Alternative command*

$$\frac{\begin{array}{l}(r)\ \{p \wedge B(g_k)\}\ C(g_k); S_k\ \{q\},\quad k = 1, \ldots, m \\ [p \wedge B(g_k)] \Rightarrow (q1_k)^{h_i}_{h_i^{\frown}\langle CP(g_k), i, p\rangle},\quad k \in IO \\ (r)\ \{q1_k\}\ S'_k\ \{q\}, k \in IO\end{array}}{(r)\ \{p\}\ [\Box(k = 1, \ldots, m) g_k \rightarrow S_k \langle\langle S'_k \rangle\rangle]\ \{q\}}$$

where $B(g_k)$ is the boolean part and $C(g_k)$ the communication part of the guard g_k ($C(g_k)$ is *skip* if g_k is purely boolean), IO is the set of indices of the input/output guards, and $CP(g_k)$ is the index of the process with which P_i is trying to communicate in $C(g_k)$. $\langle\langle S'_k \rangle\rangle$ will be present only if g_k is an input or output guard.

R8. *Repetitive command*

$$\frac{(r)\ \left\{p \wedge \bigvee_{k=1}^{m} B(g_k)\right\}\ [\Box(k = 1, \ldots, m) g_k \rightarrow S_k \langle\langle S'_k \rangle\rangle]\ \{p\}}{(r)\ \{p\} * [\Box(k = 1, \ldots, m) g_k \rightarrow S_k \langle\langle S'_k \rangle\rangle\ \left\{p \wedge \bigwedge_{k=1}^{m} \neg B(g_k)\right\}}$$

To simplify the presentation we assume that a loop terminates only when the boolean parts of all the guards evaluate to false.

R9. *Consequence*

$$\frac{r1 \Rightarrow r,\ p \Rightarrow p1,\ (r1)\ \{p1\}\ S\ \{q1\},\ q1 \Rightarrow q}{(r)\ \{p\}\ S\ \{q\}}$$

6. Failure of a process

We now need to see how to obtain the behaviour of the failure prone execution of a process P_i from the rules given above. Let $[P_i]$ denote such an execution of P_i; $[P_i]$ then consists of a series of partial executions of P_i which end in failure and re-making of P_i, followed by a final and complete execution. (Note that any state reachable in $[P_i]$ is a state that is reachable in some execution of P_i.) The behaviour of $[P_i]$ can be defined by a rule.

R10. *Failure prone process execution*

$$\frac{\begin{array}{l}(r)\ \{p\}\ P_i\ \{q\} \\ q \Rightarrow q' \\ r \Rightarrow r' \\ [q' \wedge r^{h_i}_{h_i}] \Rightarrow q'^{h_i}_{h_i^{\frown}\langle i, 0, \delta\rangle^{\frown} h_i} \\ [r' \wedge r^{h_i}_{h_i}] \Rightarrow r'^{h_i}_{h_i^{\frown}\langle i, 0, \delta\rangle^{\frown} h_i}\end{array}}{(r')\ \{p\}\ [P_i]\ \{q'\}}$$

11006

11007

Note that in general, h_i in $r^{h_i}_{h_i'}$ and $q'^{h_j'}_{h_i'\langle i,0,\delta\rangle^\frown h_i}$ does not refer to the same sequence. This is because all predicates involved in the annotation of process P_i are described in terms of a general but arbitrary sequence named h_i. We could have written the fourth clause of R10 as

$$[q'(h_i) \wedge r(h_i')] \Rightarrow q'(h_i' \hat{} \langle i, 0, \delta\rangle \hat{} h_i)$$

where $t(H)$ would be defined to be true if and only if the predicate t is satisfied for the sequence H. We have not adopted this notation since it would make the presentation of most of the other rules much more complicated.

The second clause of R10, $q \Rightarrow q'$, ensures that the results of the executions of $[P_i]$ that proceed to completion without encountering a failure satisfy $[P_i]$'s post-condition. Similarly, the third clause of R10, $r \Rightarrow r'$, ensures that those executions of $[P_i]$ that proceed without encountering a failure satisfy $[P_i]$'s execution invariant. The fourth implication ensures that the results of those executions of $[P_i]$ that encounter $n+1$ failures before going through one fault free execution will satisfy the post-condition q' provided the results of those executions of $[P_i]$ that encounter n failures satisfy q'. This may be seen as follows.

Consider an execution of $[P_i]$ which encounters $n+1$ failures, after each of which the process is re-made with its variables set to their initial values and $\langle i, 0, \delta\rangle$ concatenated to h_i. When the final execution begins, the variables of P_i will have their initial values and the value of h_i will have the form

$$h_i^1 \hat{} \langle i, 0, \delta\rangle \hat{} h_i^2 \hat{} \langle i, 0, \delta\rangle \hat{} \cdots \hat{} h_i^{n+1} \hat{} \langle i, 0, \delta\rangle$$

where h_i^j is the record of communications that the process goes through during its jth partial execution. Let h_i'' be the sequence of communications by the process during its final execution, and let the final state of the local variables of the process be denoted by S_i^f. Since after each failure, the local state of $[P_i]$ is reset to its initial value, a possible execution of $[P_i]$ would be one in which n (rather than $n+1$) failures were encountered, with the $n+1$st execution proceeding without failure and reaching the same final state S_i^f, and with its communication sequence being

$$h_i^2 \hat{} \langle i, 0, \delta\rangle \hat{} \cdots \hat{} h_i^{n+1} \hat{} \langle i, 0, \delta\rangle \hat{} h_i''.$$

Thus the n-failure execution of $[P_i]$ is identical to the $n+1$-failure execution except that it avoids the first failure of that execution. Also, h_i^1 will satisfy r (i.e., $r^{h_i}_{h_i'}$ will be true if $h_i' = h_i^1$). Then if the results obtained following the n-failure execution of $[P_i]$ satisfy q', and the second implication in R10 is true, the results obtained following a $n+1$-failure execution will also satisfy q'. The interpretation of the fifth clause of R10 is similar.

A final remark is in order before concluding this section: it would appear that it should be possible to obtain the predicate r directly from the partial correctness post-condition q of P_i, rather than by building it up during the proof of P_i. One way of doing this would be to define

$$r \equiv \exists h_i' . [h \subseteq h_i' \wedge q^{h_i}_{h_i'}]$$

and to argue that any sequence h_i that satisfies r must be the initial subsequence of some sequence h_i' that will satisfy q. Recall, however, the example given earlier which indicates why a simple Hoare-style rule is inadequate for proving properties of fault tolerant programs. If P_5 starts execution with the pre-condition $h_5 = \varepsilon$, and no faults occur, we can obtain the post-condition $h_5 = \langle\langle 5, 6, 2\rangle\rangle$. r could then be

$$r \equiv [h_5 = \varepsilon \vee h_5 = \langle\langle 5, 6, 2\rangle\rangle].$$

Using this to derive the post-condition when faults do occur would give

$$[\forall k, 1 \leq k \leq |h_5| \,.\, [h_5[k] = \langle 5, 6, 2\rangle \vee h_5[k] = \langle 5, 0, \delta\rangle]].$$

This is incorrect, as some execution of P_5 may choose the first guard, send 1 to P_6, fail, recover (i.e. send δ to P_0), choose the next guard, send 2 to P_6 and terminate. For such an execution of P_5, we will have

$$h_5 = \langle\langle 5, 6, 1\rangle\langle 5, 0, \delta\rangle\langle 5, 6, 2\rangle\rangle$$

which does not satisfy the above post-condition. The problem, of course, is with the invariant for P_5; the right invariant for P_5 is r_5, where

$$r_5 \equiv [h_5 = \varepsilon \vee h_5 = \langle\langle 5, 6, 1\rangle\rangle \vee h_5 = \langle\langle 5, 6, 2\rangle\rangle]$$

and using the rules presented in the last section we can prove

$$(r_5)\ \{h_5 = \varepsilon\}\ P_5\ \{h_5 = \langle\langle 5, 6, 2\rangle\rangle\}.$$

Using the rule of the current section, we can then prove

$$(r_5')\ \{true\}\ [P_5]\ \{q_5'\},$$

where

$$r_5' \equiv [h_5 \in \{\langle 5, 6, 1\rangle\langle 5, 0, \delta\rangle, \langle 5, 6, 2\rangle\langle 5, 0, \delta\rangle\}^*\{\langle 5, 6, 1\rangle, \langle 5, 6, 2\rangle, \varepsilon\}]$$

and

$$q_5' \equiv [h_5 \in \{\langle 5, 6, 1\rangle\langle 5, 0, \delta\rangle, \langle 5, 6, 2\rangle\langle 5, 0, \delta\rangle\}^*\langle 5, 6, 2\rangle].$$

7. Parallel composition of fault tolerant processes

R11. *Parallel composition*

$$\frac{(r_i')\ \{p_i \wedge h_i = \varepsilon\}\ [P_i]\ \{q_i'\},\quad i = 1, \ldots, n}{\left(\bigwedge_{i=1}^{n} r_i' \wedge Compat(h_1, \ldots, h_n)\right)\ \left\{\bigwedge_{i=1}^{n} p_i\right\}\ [P_1 \| \cdots \| P_n]\ \left\{\bigwedge_{i=1}^{n} q_i' \wedge Compat(h_1, \ldots, h_n)\right\}}$$

The interesting aspect of this rule is that once the properties of fault prone processes, $[P_1], \ldots, [P_n]$ are determined, they can be used to infer the property of their composition $[P_1 \| \cdots \| P_n]$ without referring back to the processes $P_1, \ldots, P_n$. This makes the proof system compositional in nature.

The definition of *Compat* is slightly different from that of [8], to take care of the δ and ρ type elements that may appear in communication sequences:

$$Compat\ (h_1, \ldots, h_n) \equiv \exists h\ .\ [\forall i, 1 \leq i \leq n,\ [h|_r i = h_i \wedge [\forall j, 1 \leq j \leq n, i \neq j, R(h|_r i, j)]]]$$

where

$h|_r i \equiv$ the subsequence of all elements of h in which P_i participates, i.e. actual communications between P_j and P_i (i.e. elements of the form $\langle j, i, m\rangle$ or $\langle i, j, m\rangle$, where m is not ρ or δ), failure/recovery of P_i (i.e. elements of the form $\langle i, 0, \delta\rangle$), and observations by P_i of a recovery by P_j (i.e. elements of the form $\langle j, i, \rho\rangle$), but *not* observations by P_j of a recovery of P_i.

$h|_r i, j \equiv$ the subsequence of all elements of h which correspond to communication between P_i and P_j, and P_i and P_0.

Informally,

$R(h|_r i, j) \equiv P_0$ informed of P_i's faults and P_j informed of P_i's faults.

Formally,

$$R(h) \equiv \forall k\ .\ 1 \leq k \leq |h|.$$
$$[(h[k] = \langle i, j, \rho\rangle \Rightarrow h[k-1] = \langle i, 0, \delta\rangle)$$
$$\wedge [(h[k] = \langle i, 0, \delta\rangle \wedge k < |h|)$$
$$\Rightarrow [h[k+1] = \langle i, j, \rho\rangle \vee h[k+1] = \langle i, 0, \delta\rangle]]]$$

The definition ensures that if P_j has registered failure and recovery of P_i (recorded by $\langle i, j, \rho\rangle$ as the kth element), P_i must indeed have failed and recovered (recorded by a $\langle i, 0, \delta\rangle$ as the $k-1$th element). Similarly, it ensures that if P_i fails (recorded by a $\langle i, 0, \delta\rangle$), then before P_i and P_j can communicate with each other, P_j must 'register' that P_i had failed and recovered (recorded by a $\langle i, j, \rho\rangle$). Note that even when a process P_i fails several times before communicating with some other process P_j, exactly one triple $\langle i, j, \rho\rangle$ will be appended to h_j when P_j next attempts to communicate with P_i. As an example, if P_1 fails and then sends a value 3 to P_2 the various sequences will be as follows:

$h = \langle\langle 1, 0, \delta\rangle\langle 1, 2, \rho\rangle\langle 1, 2, 3\rangle\rangle$,

$h|_r 1 = h_1 = \langle\langle 1, 0, \delta\rangle\langle 1, 2, 3\rangle\rangle$,

$h|_r 2 = h_2 = \langle\langle 1, 2, \rho\rangle\langle 1, 2, 3\rangle\rangle$,

$h|_r 1, 2 = \langle\langle 1, 0, \delta\rangle\langle 1, 2, \rho\rangle\langle 1, 2, 3\rangle\rangle$,

$h|_r 2, 1 = \langle\langle 1, 2, \rho\rangle\langle 1, 2, 3\rangle\rangle$.

In effect, R11 specifies that if r_i' is the invariant of $[P_i]$ and q_i' its post-condition $(i = 1, \ldots, n)$, then the assertion $r_1' \wedge \cdots \wedge r_n'$ will hold during the execution of

$[P_1 \| \cdots \| P_n]$, since these invariants r_i' have been shown to hold during the execution of $[P_i]$ $(i = 1, \ldots, n)$, independently of what $[P_j]$, $j \neq i$, might do. Moreover, the communications recorded in the individual sequences must be mutually compatible at all times. Also when $[P_1 \| \cdots \| P_n]$ finishes, not only will the post-conditions $q_1', \ldots, q_n'$ all hold, but also *Compat* (expressing the mutual compatibility of the communications recorded in the various sequences). Thus this rule is quite similar to the parallel composition rule of [8].

8. Non-terminating processes and programs

The axioms and rules of the last three sections can be used to obtain the post-condition of fault tolerant programs; they can also be used to deal with non-terminating programs, such as the bounded buffer program given earlier. In this section we explain how we can use the approach of this paper to deal with such programs.

Consider a fault tolerant program

$$[P_1 \| \cdots \| P_n]$$

Suppose for each of the processes P_i $(i = 1, \ldots, n)$ we have shown, using the axioms and rules of the last section, the following results:

$$(r_i')\ \{p_i\}\ [P_i]\ \{q_i'\}$$

If P_i is a non-terminating process, q_i' will presumably be the predicate *false*; however, we are interested not in the post-condition of P_i (or of the other processes), but in the predicates it satisfies at certain points during its execution. Frequently, for instance, we are interested in the invariant relation that is satisfied by the variables of P_i (in particular by its communication sequence h_i) at all times during P_i's execution. If each of the processes is non-terminating, as is often the case (as for the bounded buffer), we are usually interested in the invariant relation that the communication sequences of the various processes satisfy; in the case of the bounded buffer we would probably like to prove that the lines produced by P_1 are received by P_4 in proper order, possibly with some repetitions, and that the number of repetitions is bounded by some function of the number of failures of the processes P_2 and P_3.

The rule R11 of the last section allows us easily to arrive at the invariant

$$r_1' \wedge r_2' \wedge \cdots \wedge r_n' \wedge \mathit{Compat}(h_1, \ldots, h_n).$$

In the next section, we show that the bounded buffer program does indeed behave as we expect it to. The reader may recall that in the bounded buffer program we assumed that P_1 and P_4 do not fail; clearly, then, if we can prove

$$(r_i)\ \{h_i = \varepsilon\}\ P_i\ \{q_i\}, \quad i = 1, \ldots, 4,$$

the appropriate invariant for the whole program, allowing for faults in P_2 and P_3 but not in P_1 and P_4, would be

$$r_1 \wedge r_2' \wedge r_3' \wedge r_4 \wedge Compat(h_1, \ldots, h_4)$$

rather than

$$r_1' \wedge r_2' \wedge r_3' \wedge r_4' \wedge Compat(h_1, \ldots, h_4).$$

In the next section we shall prove

$$(r_i) \ \{h_i = \varepsilon\} \ P_i \ \{false\}, \quad i = 1, \ldots, 4$$

with appropriate r_i (the post-condition is identically *false* as each of the processes is in an infinite loop), and show that

$$[r_1 \wedge r_2' \wedge r_3' \wedge r_4 \wedge Compat(h_1, \ldots, h_4)] \Rightarrow [T_1 \wedge T_2 \wedge T_3 \wedge T_4] \tag{3}$$

where

$$T_1 \equiv \forall k \,.\, 1 \leq k \leq |h_4| \,.\, \{h_4[k] \in \{\langle 3, 4, \rho\rangle, \langle 3, 4, Line(m)\rangle\}\}, \tag{4}$$

that is

$$h_4 \in \{\langle 3, 4, \rho\rangle, \langle 3, 4, Line(m)\rangle\}^*.$$

Thus T_1 in (3) will essentially show that the only 'normal' values the printer process P_4 will receive are the 'lines'. (Recall that P_4 prints all the normal values it receives.)

$$\begin{aligned} T_2 \equiv\ & [\forall k \,.\, 1 \leq k \leq |h_4| \,.\, \{h_4[k] = \langle 3, 4, \rho\rangle\} \\ & \vee [\exists k \,.\, 1 \leq k \leq |h_4| \,.\, \{h_4[k] = \langle 3, 4, Line(1)\rangle \\ & \qquad \wedge \forall k' \,.\, 1 \leq k' < k \,.\, \{h_4[k'] = \langle 3, 4, \rho\rangle\}\}], \end{aligned} \tag{5}$$

that is

$$h_4 \in \{\langle 3, 4, \rho\rangle\}^*$$

or

$$h_4 \in \{3, 4, \rho\rangle\}^* \langle 3, 4, Line(1)\rangle \{3, 4, \rho\rangle, \langle 3, 4, Line(m)\rangle\}^*.$$

T_2 ensures that the first normal value that P_4 receives is $Line(1)$

$$\begin{aligned} T_3 \equiv\ & \forall k, k' \,.\, 1 \leq k \leq k' \leq |h_4|. \\ & \{h_4[k] = \langle 3, 4, Line(m)\rangle \wedge h_4[k'] = \langle 3, 4, Line(m')\rangle \\ & \Rightarrow \forall m'' \,.\, m < m'' < m'. \\ & \{\exists k'' \,.\, k' < k'' < k \,.\, \{h_4[k''] = \langle 3, 4, Line(m'')\rangle\}\}\}. \end{aligned} \tag{6}$$

T_3 ensures that if at some time t the mth line is printed and at a later time t' the m'th line is printed then all lines from $m+1$ to $m'-1$ are printed between time t and t' (possibly with duplications).

$$T_4 \equiv f_2(h_4) \leq (n-1) * f_1(h_2) \tag{7}$$

where

$f_1(h_2)$ = number of elements of the kind $\langle 2, 0, \delta \rangle$ in h_2, i.e., the number of failures of P_2,

$f_2(h_4)$ = number of repetitions in elements of the kind $\langle 3, 4, Line(m) \rangle$ in h_4.

$f_1(h_2)$ and $f_2(h_4)$ can be formally defined in a straightforward manner and we leave that for the reader. Thus T_4 specifies an upper bound on the number of duplications in the lines printed; it is possible to get a tighter bound involving h_2, h_3 and h_4; however, this would be much more complex than T_4, since it would involve the relative *times* at which P_2 and P_3 failed (not just the *number* of failures of P_2 and P_3).

With T_1, T_2, T_3 and T_4 as defined above, (3) will clearly show that the program does indeed behave as expected.

9. Proof of the bounded buffer program

Our proof of the bounded buffer program will be quite informal; we shall begin by proving the results

$$(r_i) \quad \{h_i = \varepsilon\} \quad P_i \quad \{false\}, \quad i = 1, \ldots, 4. \tag{8}$$

The formalization of these proofs (appealing to the various axioms and rules applicable to the statements in P_i) will be left to the interested reader. Our informal arguments will be rather like the semi-formal proofs of sequential programs, omitting most of the intermediate details and all but the key assertions such as loop invariants; such informal proofs are justified since our formalism allows (or rather requires)us to consider one process at a time, and the validity of the various assertions in the proof of a process depends entirely on what the process does—and not at all on what the other processes do or on how they interact with this process or with each other. Thus once the intuition behind the proof rules is understood, it is as easy to informally prove results such as (8) as it is to informally prove the partial correctness of sequential programs.

However, in the current case, we have to prove an additional result, after proving (8):

$$[r_1 \wedge r_2' \wedge r_3' \wedge r_4 \wedge Compat(h_1, \ldots, h_4)] \Rightarrow [T_1 \wedge T_2 \wedge T_3 \wedge T_4].$$

Let us begin by considering process P_4 since it is the simplest. The reader should easily be able to see the following result:

$$(r_4) \quad \{h_4 = \varepsilon\} \quad P_4 \quad \{false\} \tag{9}$$

where

$$r_4 \equiv \forall k \,.\, 1 \leq k \leq |h_4| \,.\, \{h_4[k] \in \{\langle 3, 4, m \rangle, \langle 3, 4, \rho \rangle\} \wedge m \notin \{\rho, \delta\}\}, \tag{10}$$

that is

$$h_4 \in \{\langle 3, 4, m \rangle, \langle 3, 4, \rho \rangle\}^*$$

(the loop invariant will be identical to r_4).

The post-condition in (9) merely expresses the fact that the loop does not terminate; r_4 will be true at all times during the execution of P_4 since the only communications P_4 participates in are those in which it receives an input from P_3 or a signal that P_3 has failed and recovered since the previous communication between P_3 and P_4.

Next consider P_1; the loop invariant for it is:

$$\begin{aligned} LI_1 \equiv\ & h_1 \in \{\langle 1, 2, \rho\rangle, \langle 1, 2, (Line(m), m)\rangle\}^* \\ & \wedge\ ready \Rightarrow nextline = Line(pseq + 1) \\ & \wedge\ pseq = g_1(h_1) \wedge sent = g_2(h_1) \\ & \wedge \forall k \,.\, 1 \leq k \leq |h_1| \,.\, [h_1[k] = \langle 1, 2, (p, q)\rangle \\ & \qquad \Rightarrow p = Line(q) \wedge q = g_1(h_1[1 : k-1]) + 1] \end{aligned}$$

where

$$\begin{aligned} & g_1(\varepsilon) = 0, && g_2(\varepsilon) = 0 \\ & g_1(h \hat{}\ \langle 1, 2, (p, q)\rangle) = g_1(h) + 1, && g_2(h \hat{}\ \langle 1, 2, (p, q)\rangle) = g_2(h) + 1, \\ & g_1(h \hat{}\ \langle 1, 2, \rho\rangle) = g_1(h) - min(n, g_2(h)), && g_2(h \hat{}\ \langle 1, 2, \rho\rangle) = 0. \end{aligned}$$

The relation r_1 is

$$\begin{aligned} r_1 \equiv\ & \forall k \,.\, 1 \leq k \leq |h_1| \,. \\ & [h_1[k] = \langle 1, 2, \rho\rangle \\ & \vee\ h_1[k] = \langle 1, 2, (Line(g_1(h_1[1 : k-1]) + 1), g_1(h_1[1 : k-1]) + 1)\rangle]. \end{aligned}$$

For readability and simplicity we will instead use the following relation for r_1:

$$\begin{aligned} r_1 \equiv\ & \forall k \,.\, 1 \leq k \leq |h_1| \,.\, [h_1[k] = \langle 1, 2, (p, q)\rangle \\ & \qquad \Rightarrow p = Line(q) \wedge q = g_1(h_1[1 : k-1]) + 1]. \end{aligned}$$

As stated earlier, we will leave it to the reader to formally verify $(r_1)\{h_1 = \varepsilon\}\ P_1\ \{false\}$, using LI_1 as the loop invariant, and introducing appropriate assertions as necessary.

Next consider P_2 for which we will only specify r_2, leaving the other assertions including the loop invariant to the reader:

r_2 = what P_2 receives from P_1 is sent out to P_3
and the number of values received from P_1 but not sent to P_3 can be at most $n-1$.

More formally,

$$\begin{aligned} r_2 \equiv\ & [f_3(h_2|3)]_{1,2}^{2,3} \subseteq h_2|1 \\ & \wedge\ [|f_3(h_2|3)| \leq |h_2|1| \leq |f_3(h_2|3)| + n - 1] \end{aligned}$$

where

$f_3(h_2|3)$ = sequence obtained from $h_2|3$ by dropping from it all subsequences of the form $\langle\langle 2, 3, m\rangle\langle 3, 2, \rho\rangle\rangle$ for all m

= $h_2|3$ if in that execution there were no failures in P_3

= $h_2|3$ in the execution of P_2 where the fault alternative $\langle\langle \cdots \rangle\rangle$ has been removed,

$[h]^{x_1,\ldots,x_n}_{e_1,\ldots,e_n}$ = sequence obtained from h by replacing simultaneously all occurrences of x_i by e_i for $i = 1, \ldots, n$.

More formally $f_3(h_2|3)$ is defined as follows:

$$f_3(\varepsilon) = \varepsilon, \qquad f_3(\langle 3, 2, \rho\rangle \hat{} h) = f_3(h),$$

$$f_3(\langle 2, 3, m\rangle \hat{} \langle 2, 3, m'\rangle \hat{} h) = \langle 2, 3, m\rangle \hat{} f_3(\langle 2, 3, m'\rangle \hat{} h) \quad \text{for } m, m' \neq \rho,$$

$$f_3(\langle 2, 3, m\rangle \hat{} \langle 3, 2, \rho\rangle \hat{} h) = f_3(h).$$

In writing down r_2, we have allowed for failures in P_3, but not in P_1 since P_1 is assumed not to fail; failures in P_2 will be accounted for when we write down r_2'. We still need to consider P_3. Again, we will only specify r_3, leaving the formal verification of (r_3) $\{h_3 = \varepsilon\}$ P_3 $\{false\}$ to the reader. First define

$f_4(\{(*, *, (m_1, m_2))\}^*, i)$ = retain only those triples $\langle *, *, m_1\rangle$ whose second component values are in increasing order starting greater than i.

Note that the difference between P_3' and the fault tolerant P_3 given above is that some 'evasive action' is taken in the fault tolerant verison to compensate for possible failures in P_2. Then, $f_4(h_3|2, 0)$ of fault tolerant $P_3 = h_3|2$ of P_3':

$$r_3 \equiv [h_3|4]^{3,4}_{2,3} \subseteq f_4(h_3|2, 0)$$
$$\wedge\, [|h_3|4| \leq |f_4(h_3|2, 0)| \leq |h_3|4| + 1].$$

More formally

$$f_4(\varepsilon, i) = \varepsilon, \qquad f_4(\langle 2, 3, \rho\rangle \hat{} h, i) = f_4(h, i),$$

$$f_4(\langle 2, 3, (m_1, m_2)\rangle \hat{} h, i) = \text{if } m_2 \leq i \text{ then } f_4(h, i)$$
$$\text{else } \langle 2, 3, m_1\rangle \hat{} f_4(h, m_2).$$

r_3 expresses the fact that $[h_3|4]^{3,4}_{2,3}$ is a prefix of $f_4(h_3|2, 0)$ and that the length of $h_3|4$ can be at most 1 less than that of $f_4(h_3|2, 0)$.

That completes the informal proofs of

$$(r_i)\ \ \{h_i = \varepsilon\}\ \ P_i\ \ \{false\}, \quad i = 1, \ldots, 4$$

Before trying to prove

$$[r_1 \wedge r_2' \wedge r_3' \wedge r_4 \wedge Compat(h_1, \ldots, h_4)] \Rightarrow [T_1 \wedge T_2 \wedge T_3 \wedge T_4] \tag{11}$$

we write down r_2' and r_3' (from r_2 and r_3 respectively):

$$r_2' \equiv \forall k \,.\, 1 \leq k \leq Num(h_2, \delta) + 1 \,.\, \{r_2^{h_2_{subseq(h_2, k-1, k, \delta)}}\}$$

where

$Num(h_i, x)$ = number of triples of the kind $\langle j, i, x\rangle$ in h_i,

$subseq(h_i, m1, m2, x)$ = the subsequence of h_i from just after the $m1$th element of h_i of the form $\langle j, i, x\rangle$ (from the beginning of h_i if $m1 = 0$) to just before the $m2$th element of h_i of the form $\langle j, i, x\rangle$ (to the end of h_i if $m > Num(h_i, x)$).

Essentially r_2' says that h_2 looks like a concatenation of a number of (smaller) sequences each of which satisfies r_2, with a $\langle 2, 0, \delta\rangle$ element sandwiched between each consecutive pair of these smaller sequences. r_3' is similar:

$$r_3' \equiv \forall k \,.\, 1 \leq k \leq Num(h_3, \delta) + 1 \,.\, \{r_3^{h_3_{subseq(h_3, k-1, k, \delta)}}\}.$$

Proof of T_1. From r_1 and r_2' we can infer that

$$h_2 | 1 \in \{\langle 1, 2, (Line(m), m)\rangle\}^*.$$

This together with r_3' gives us

$$h_3 | 2 \in \{\langle 2, 3, (Line(m), m\rangle, \langle 2, 3, \rho\rangle\}^*$$

Combining this with r_4 we get

$$h_4 \in \{\langle 3, 4, Line(m)\rangle, \langle 3, 4, \rho\rangle\}^*. \quad \square$$

Proof of T_2. From r_1 and r_2' we know that

$$h_2 | 3 \in \{\langle 2, 3, (Line(1), 1)\rangle\langle 3, 2, \rho\rangle, \langle 3, 2, \rho\rangle\}^*$$
$$\{\langle 2, 3, (Line(1), 1)\rangle\langle 2, 3, (Line(2), 2)\rangle\}\ \hat{}\ \text{some arbitrary trace.}$$

Combining this with r_3' gives us

$$h_3 | 4 \in \langle 3, 4, Line(1)\rangle\ \hat{}\ \text{some arbitrary trace}$$

and hence

$$h_4 \in \{\langle 3, 4, \rho\rangle\}^*\langle 3, 4, Line(1)\rangle\ \hat{}\ \text{some arbitrary trace.} \quad \square$$

Proof of T_3. From r_1 and r_2' we obtain

$$\forall k \,.\, 1 \leq k \leq |h_2| \,.\, \{h_2[k] = \langle 1, 2, (Line(m), m)\rangle \wedge m > 1$$
$$\Rightarrow \exists k' \,.\, k' < k \,.\, \{h_2[k'] = \langle 1, 2, (Line(m-1), m-1)\rangle$$
$$\wedge\ h_2[k'+1 : k-1] | 1 \in \{\langle 1, 2, (Line(p), p)\rangle\}^*$$
$$\text{where } p \geq m\}\}.$$

Combining this with r_2' gives us

$$\forall k \,.\, 1 \leq k \leq |h_3| \,.\, \{h_3[k] = \langle 2, 3, (Line(m), m)\rangle \wedge m > 1$$
$$\Rightarrow \exists k' \,.\, k' < k \,.\, \{h_3[k'] = \langle 2, 3, (Line(m-1), m-1)\rangle$$
$$\wedge\, h_3[k'+1 : k-1] \,|\, 2 \in \{\langle 2, 3, (Line(p), p)\rangle, \langle 2, 3, p\rangle\}^*$$
$$\text{where } p \geq m\}\}.$$

From the above and r_4 we can infer that

$$\forall k \,.\, 1 \leq k \leq |h_4| \,.\, \{h_4[k] = \langle 3, 4, Line(m)\rangle \wedge m > 1$$
$$\Rightarrow \exists k' \,.\, k' < k \,.\, \{h_4[k'] = \langle 3, 4, Line(m-1)\rangle$$
$$\wedge\, h_4[k'+1 : k-1] \in \{\langle 3, 4, Line(p)\rangle, \langle 3, 4, p\rangle\}^*$$
$$\text{where } p \geq m\}\}.$$

which can be rewritten as

$$\forall k, k' \,.\, 1 \leq k \leq k' \leq |h_4|.$$
$$\{h_4[k] = \langle 3, 4, Line(m)\rangle \wedge h_4[k'] = \langle 3, 4, Line(m')\rangle$$
$$\Rightarrow \forall m'' \,.\, m < m'' < m' \,.\, \{\exists k'' \,.\, k' < k'' < k$$
$$\,.\, \{h_4[k''] = \langle 3, 4, Line(m'')\rangle\}\}\}. \quad \square$$

Proof of T_4. From r_1 we obtain

$$\text{number of repetitions in } h_1|2 \leq (n-1) * \text{number of failures of } P_2.$$

Combining this with r_2' we get

$$\text{number of repetitions in } h_2|3 \leq (n-1) * \text{number of failures of } P_2$$
$$+ \text{number of failures of } P_3.$$

From the above and r_3' we get

$$\text{number of repetitions in } h_3|4 \leq (n-1) * \text{number of failures of } P_2.$$

Since

$$\text{number of repetitions in } h_3|4 = \text{number of repetitions in } h_4$$

we have proved T_4. $\square$

10. Discussion

The proof rules defined here provide a means for formally proving the properties of fault tolerant programs written in extended CSP and executing on a distributed system whose nodes may fail. They are naturally more complex than proof rules for programs that execute on totally reliable hardware but, with some familiarity, their interpretation becomes no more complicated than the task of constructing fault

tolerant programs. The difficulty of writing correct, fault tolerant programs is easily underestimated and, during the course of this work we found that the exercise of conducting formal proofs often revealed errors in programs that earlier passed through fairly thorough but informal inspection.

The proof system proposed in this paper is compositional in nature; that is it is possible to derive the specification of a construct purely from the specification of its components.

A criticism that may be directed at our proof system is that the parallel composition rule combines all the processes at once; rather than, say, two at a time. This is not so much due to our formalism, as it is to the CSP framework in which one cannot combine two processes to obtain a new process. It is possible, using the same intuition as that underlying the system of this paper, to construct a proof system in which processes can be bound together two at a time if that is reasonable in the underlying framework. Since binding processes in pairs does not seem reasonable in CSP, we did not attempt to do this in the current paper.

The rules are limited to proofs of partial correctness and it is tempting to consider how they may be extended to deal with total correctness. For example, if it can be proved that all loops will terminate, the function *Compat* can be extended to detect deadlock and to prove process termination. Unfortunately, proof of loop termination cannot in general be done in isolation and requires re-inspection of the proof outlines of all processes. This militates against the basic intention of this proof system that once the proofs of individual processes are over, proof of the program should directly result from the application of the rule of parallel composition. The possibility of carrying forward loop termination predicates in post-conditions and proving their 'compatibility' at the end must be rejected because it results in extremely unwieldy proofs. So the proof system remains one of partial correctness.

It is important to prove that the axiomatic system is consistent and complete with respect to an operational model for extended CSP. Though we shall not attempt to demonstrate this here, the proof given in [9] for the system of [8] can be adapted for our version of extended CSP.

Throughout this paper, we have assumed that a process executing on a failed node is restarted from its initial state. Such an assumption allows the possibility of running the system of processes using read only stable storage [5]. If checkpoints of processor states are stored, a process can be restarted from a more recent state than its initial state. This will require the usage of stable storage but it may at times simplify the recovery action required. Checkpoints can be handled by suitably altering axiom R10 for Failure Prone Process Execution.

Acknowledgment

We would like to take this opportunity to thank the referees for their numerous comments and suggestions that greatly improved the presentation of this paper.

References

[1] K.R. Apt, N. Francez and W.P. de Roever, A proof system for communicating sequential processes, *ACM TOPLAS* **2** (1980) 359-385.

[2] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978) 666-677.

[3] M. Joseph and A. Moitra, Fault tolerance in communicating processes, *Conference Record, 2nd FST & TCS Conference*, Bangalore, India, (1982); An expanded version of this appears as TR-72, NCSDCT.

[4] M. Joseph and A. Moitra, Cooperative recovery from faults in distributed programs, in: R.E.A. Mason, Ed., *Information Processing 1983* (North-Holland, Amsterdam, 1983) 481-486.

[5] B. Lampson, Atomic transactions, in: *Distributed Systems—Architecture and Implementation*, Lecture Notes in Computer Science **105** (Springer, New York, 1981) 246-265.

[6] M. Pease, R. Shostak and L. Lamport, Reaching agreement in the presence of faults, *J. ACM* **27** (1980) 228-234.

[7] R.D. Schlichting and F.B. Schneider, Failstop processors: an approach to designing fault-tolerant computing systems, *ACM Trans. Comput. Systems* **1** (1983) 222-238.

[8] N. Soundararajan, Correctness proofs of CSP programs, *Proc. 1st FST & TCS Conference*, Bangalore, India (1981) 135-142; An expanded version appears in *Theoret. Comput. Sci.* **24** (1983) 131-141.

[9] N. Soundararajan and O.J. Dahl, Partial correctness semantics of CSP, *BIT*, to appear.